Appears in the Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)

Fast Path-Based Neural Branch Prediction

Daniel A. Jiménez
Department of Computer Science
Rutgers University, Piscataway, NJ 08854
djimenez@cs.rutgers.edu

Abstract

Microarchitectural prediction based on neural learning has received increasing attention in recent years. However, neural prediction remains impractical because its superior accuracy over conventional predictors is not enough to offset the cost imposed by its high latency. We present a new neural branch predictor that solves the problem from both directions: it is both more accurate and much faster than previous neural predictors. Our predictor improves accuracy by combining path and pattern history to overcome limitations inherent to previous predictors. It also has much lower latency than previous neural predictors. The result is a predictor with accuracy far superior to conventional predictors but with latency comparable to predictors from industrial designs. Our simulations show that a path-based neural predictor improves the instructions-per-cycle (IPC) rate of an aggressively clocked microarchitecture by 16% over the original perceptron predictor.

1 Introduction

Branch misprediction latency is the most important component of performance degradation as microarchitectures become more deeply pipelined [20]. Branch predictors must improve to avoid the increasing penalties of mispredictions. Branch predictors based on neural learning are the most accurate predictors in the literature [12, 10], but they are impractical because the advantage of the extra accuracy is nullified by high access latency, even when latency-sensitive predictor organizations are used [7]. This latency is due primarily to the complex computation that must be carried out to determine the excitation of an artificial neuron.

We present a new, practical neural branch predictor. Its latency is much lower than previous designs and is comparable to that of conventional predictors used in industrial designs, making it practical for implementation in a high-frequency microprocessor. At the same time, its accuracy is superior to that of previous highly accurate predictors.

Figure 1 illustrates how our new predictor achieves low latency by beginning well ahead of time. The predictor staggers computations in time, predicting a branch using a neuron selected dynamically along the path to that branch, rather than selecting the neuron all at once based solely on the branch address. A happy side-effect of this selection process is improved accuracy because the predictor is able to correlate with path history as well as pattern history.

We show that our path-based neural predictor has a misprediction rate 7% lower than that of the original perceptron predictor, and because of its improved latency it delivers an IPC 16% higher than that predictor at a 64KB hardware budget.

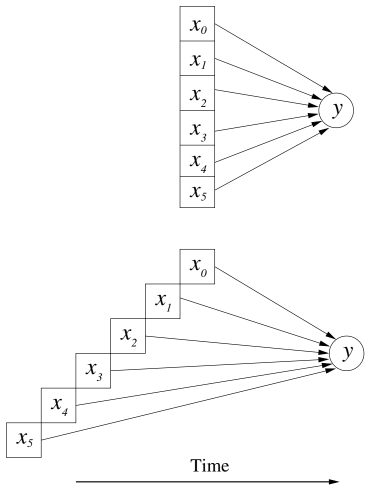

Figure 1. Rather than being done all at once (above), computation is staggered (below).

This paper is organized as follows: Section 2 briefly discusses related work. Section 3 gives background in neural branch prediction and explains the new prediction algorithm. Section 4 describes our experimental methodology. Section 5 gives the accuracy and performance results of our experiments. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Neural Prediction

Calder *et al.* use neural networks to perform static branch prediction [4] at compile time. Features such as control-flow information are used to train a neural network to distinguish between branches that are likely to be biased taken from branches that are likely to be biased not taken.

Dynamic branch prediction with neural methods was first proposed by Vintan *et al.* [23] who explore the use of learning vector quantization, a neural method. The resulting branch predictor achieves an accuracy comparable to a table-based branch predictor.

2.1.1 Branch Prediction with Perceptrons

The original perceptron predictor [9] uses a simple linear neuron known as a perceptron [1] to perform branch prediction. Perceptrons achieve better accuracy than two-level adaptive branch prediction because of their ability to exploit long history lengths which have been shown to provide additional correlation for branch predictors [6]. Another study suggests ways to implement the predictor using techniques from high-speed arithmetic [10], but the latency of the predictor is more than 4 cycles with an aggressive clock rate. Despite its drawbacks, neural prediction has been suggested as a promising technology for future microprocessors [16]. It has become part of one of Intel's IA-64 simulators for researching future microarchitectures [2]. It has been used as a component in studies of hybrid predictors [12, 22] and is the most accurate single-component branch predictor in the literature [12, 10].

2.2 Path-Based Prediction

Our path-based neural predictor achieves superior accuracy and low latency by choosing the neural weights based on the path taken to reach a branch rather than the branch address itself. Branch outcomes are highly correlated both with path and pattern histories [14, 21]. Previous work has also explored the use of path information to improve branch predictor accuracy.

2.3 Latency-Sensitive Prediction

As hardware budgets for branch predictors expand, research has begun to focus on balancing the tradeoff between accuracy and latency important for large predictors with high latencies. Jiménez *et al.* survey several techniques for mitigating branch predictor delay [8]. The most common technique is *overriding*, in which a quick but relatively inaccurate predictor guides instruction fetch in a single cycle, and may be corrected by a slower but more accurate multi-cycle predictor. This approach was used for the Alpha EV6 and EV7 cores [11] and was proposed for the Alpha EV8 [16]. The overriding technique does not scale well as branch predictor latency increases because the penalty for an overriding event becomes substantial [7].

Other studies propose pipelined branch predictors [21, 7, 17] to mitigate latency. The main source of latency for most large branch predictors is the access delay to the memories used to implement the pattern history tables. The latency of the perceptron predictor is dominated by computation time.

3 A Path-Based Neural Predictor

In this section, we review the relevant details of previous work on neural branch prediction. In this context, we give the intuition behind the path-based neural predictor. We then give a detailed explanation of the path-based neural predictor.

3.1 Branch Prediction with Perceptrons

The perceptron predictor uses perceptron learning [15, 1] to predict the directions of conditional branches [9, 10]. We review the design of the perceptron predictor, describing algorithms using an Algol-like pseudocode with keywords in **boldface** and comments in *italics*. We use *taken* and *not_taken* as meaningful names for Boolean constants.

The perceptron predictor is similar to other predictors in that it keeps a global history shift register that records the outcomes of branches as they are executed, or speculatively as they are predicted. The width of this register is the history length for the predictor, hereafter referred to as h.

The perceptron predictor keeps an $n \times (h+1)$ matrix $W[0..n-1, 0..h]$ of integer *weights*, where n is a design parameter. Weights are typically 8-bit bytes. Each row of the matrix is an $(h+1)$-length *weights vector*. Each weights vector stores the weights of one perceptron that is controlled by perceptron learning. In a weights vector $w[0..h]$, the first weight, $w[0]$, is known as the *bias weight*. Thus, the first column of W contains the bias weights of each weights vector. The Boolean vector $G[1..h] \in \{1..h\} \times \{\mathit{taken}, \mathit{not\_taken}\}$ represents the global history shift register.

3.1.1 Prediction and Update Algorithms

Figure 2 gives pseudocode for the prediction and update algorithms for the original perceptron predictor. The *prediction* algorithm returns a Boolean value predicting the branch at address *pc*.

When a branch outcome becomes known, the *train* algorithm is invoked to update the predictor. The training algorithm takes an integer parameter θ that controls the tradeoff between long-term accuracy and the ability to adapt to phase behavior. It has been empirically determined that choosing $\theta = \lfloor 1.93h + 14 \rfloor$ gives the best accuracy [10]. Thus, θ is a constant for a given history length. Once the outcome of a branch becomes known, the following algorithm is used to update the perceptron predictor, taking as

function *prediction* (*pc*: **integer**): { *taken* , *not_taken* };	
begin	
$i := pc \bmod n$	*Hash the pc to select a row of W*
$y_{out} := W[i,0] + \sum_{j=1}^{h} \begin{cases} W[i,j] & \text{if } G[j] = \textit{taken} \\ -W[i,j] & \text{if } G[j] = \textit{not\_taken} \end{cases}$	*Compute output of i^{th} perceptron using $G[1..h]$ as input*
if $y_{out} \geq 0$ **then**	
prediction := *taken*	
else	*Make the prediction based on the sign of y_{out}*
prediction := *not_taken*	
end if	
end	
procedure *train* (i, y_{out}: **integer**; *prediction* , *outcome* : { *taken* , *not_taken* });	
if *prediction* $\neq$ *outcome* **or** $\lvert y_{out} \rvert \leq \theta$ **then**	*If incorrect or y_{out} below threshold then adjust weights*
$W[i,0] := W[i,0] + \begin{cases} 1 & \text{if } \textit{outcome} = \textit{taken} \\ -1 & \text{if } \textit{outcome} = \textit{not\_taken} \end{cases}$	*Increment bias weight if taken, decrement if not taken*
for j **in** $1..h$ **in parallel do**	
$W[i,j] := W[i,j] + \begin{cases} 1 & \text{if } \textit{outcome} = G[j] \\ -1 & \text{if } \textit{outcome} \neq G[j] \end{cases}$	*Increment j^{th} weight for positive correlation,*
end for	*decrement for negative correlation*
end if	
$G := (G << 1)$ **or** *outcome*	*Update the global history shift register*
end	

Figure 2. Perceptron prediction and update algorithm

parameters the outcome as well as the values of i, *prediction*, and y_{out} computed during the prediction phase.

3.1.2 Implementation

We review some of the suggestions for a practical implementation of the perceptron predictor.

The matrix W should be implemented as a tagless direct-mapped memory of n blocks with the i^{th} block containing h 8-bit weights that form the weights vector of the i^{th} perceptron. Thus, each time a prediction is needed, the weights vector corresponding to that value of *pc* is read from memory.

Instead of negating the weights to produce summands for the computation of y_{out}, they can be bitwise complemented with very little impact on accuracy. This speeds the computation of the summands.

The computation of y_{out} can be arranged as a Wallace-tree [5] adder to add the summands. This allows the circuit performing this computation to have a depth of $O(\log h)$ gate delays, as opposed to $O(h)$ gate delays with a naive summing algorithm.

3.1.3 Disadvantage of the Perceptron Predictor

The main disadvantage of the perceptron predictor is its high latency. Even using the high-speed arithmetic tricks mentioned above, the latency of the computation of y_{out} is high relative to the clock period of a deeply pipelined microarchitecture. It has been shown that performance is highly sensitive to high branch predictor latency [8], even when special techniques are used to mitigate latency [7].

3.2 A Path-Based Neural Predictor

Our alternative to the perceptron predictor is a neural predictor that chooses its weights vector according to the path leading up to a branch, rather than according to the branch address alone. This technique has two advantages. First, latency is mitigated because computation of y_{out} can begin in advance of the prediction, with each step proceeding as soon as a new element of the path is executed. Second, accuracy is improved because the predictor incorporates path information into the prediction.

3.2.1 Intuitive Description

Our new predictor has much the same structure as the perceptron predictor. It keeps a matrix W of weights vectors. Each time a branch is fetched and requires a prediction, one of the weights vectors from W is read. However, only the 0^{th} weight, i.e. the bias weight, is used to help predict the current branch. Its value is added to a running total that has been kept for the last h branches, with each summand added during the processing of a previous branch.

Figure 3 illustrates the difference between the perceptron predictor (a) and our new predictor (b). The diagrams show the progress of the two predictors predicting a sequence of branches labeled b_{t-7} through b_t with b_t fetched most recently. The vertical columns correspond to the rows of W

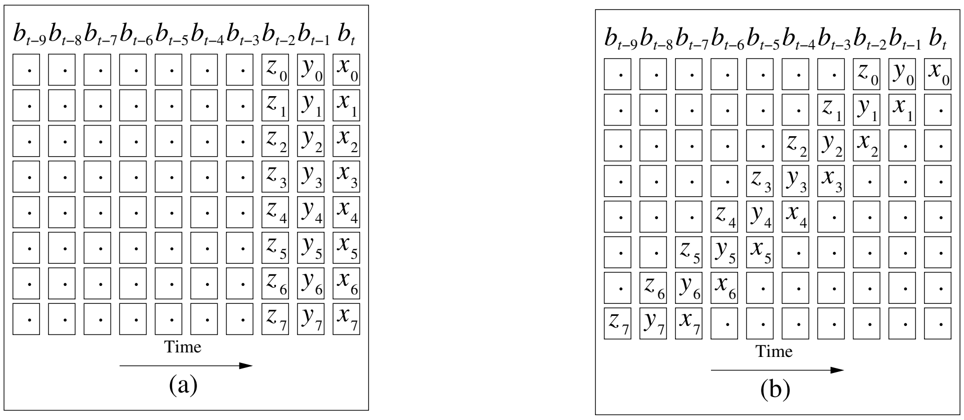

Figure 3. Illustration of the weights used to predict branch b_t with the perceptron predictor (a) and the path-based neural predictor (b) with history length of 7. Vertical columns are weights vectors.

accessed at each time step. Each predictor has a history length $h = 7$. For each predictor, the set of weights used to predict branch b_t is $x[0..7]$. For the perceptron predictor (a) at time t, the vector is accessed, each weight processed into a summand, and y_{out} computed all at once. By contrast, the $x[0..7]$ weights for the new predictor (b) are built up by accessing different positions in the weights vectors associated with branches b_{t-7} through b_t. For the new predictor, a running total (not shown) is kept of the summands in the computation of y_{out}. By time t, the only summand left to be added is the bias weight, $x[0]$. Note that the prediction generated for branch b_{t-i} is the i^{th} most recent speculative history bit for branch b_t, so the relevant parts of the speculative global history shift register become available as they are needed. To further clarify the intuition, Figure 3 also shows the positions of the weights $y[0..7]$ and $z[0..7]$ used to compute the predictions for the previous two branches b_{t-1} and b_{t-2}.

Another way to see the difference between the two predictors is to look at which weights are used to predict which branches. In the original perceptron predictor, each of the weights in the weights vector associated with branch b_j is used to predict branch b_t. In the new predictor, the i^{th} weight in the weights vector associated with branch b_j is used to predict branch b_{j+i}.

3.2.2 The Prediction Algorithm

Figure 4 shows a parallel algorithm for predicting the current branch and updating computations for predicting the next h branches. Let W, n, and h be defined as before. Let $SR[0..h]$ and $R[0..h]$ be vectors of $h+1$ integers. The first column of W form the bias weights. $SR[h-j]$ contains the running total computing the perceptron output that will be used to predict the j^{th} branch after the current one. SR is updated speculatively, so R, used in the updating algorithm described later, holds the most up-to-date non-speculative version of SR. Think of SR as a queue that holds the partial sums for the perceptron output computation as they are being computed. A zero enters the tail of the queue at $SR[0]$ and the perceptron output, minus the bias weight, emerges at $SR[h]$. SG and G are shift registers that hold speculative and non-speculative global history, respectively.

3.2.3 Update Algorithm

Updating the path-based neural predictor is conceptually similar to updating the original perceptron predictor. However, the new update algorithm has to deal with the fact that each weights vector is associated with h branches, rather than one branch as in the original predictor. When branch b_t completes and its outcome is ready to be used to update the predictor, most of the weights vector associated with b_t cannot be updated because they are being used to predict future branches that have not completed yet. Thus, we design the matrix W as $h+1$ independently addressable high-speed memories, each representing the n weights of a single column of W. When the predictor is updated the corresponding weights can be accessed independently. The memory with the bias weights are kept closest to the logic that computes the final y_{out} value for low latency.

Figure 5 gives a parallel algorithm for updating the path-based neural predictor. It accepts as parameters the values of i and y_{out} computed during the prediction algorithm as well as the Boolean outcome of the branch, a vector H representing the value of the speculative global history shift register SR when the branch was predicted, and an array $v[1..h]$ of integers representing the addresses of the last h

```
function prediction (pc: integer) : { taken , not_taken }
begin
    i := pc mod n                         Hash pc to produce i, a row number in W.
    y := SR[h] + W[i, 0]                  y, the perceptron output, is the partial sum
    if y ≥ 0 then                         from the last prediction plus the bias weight.
        prediction := taken
    else                                  The prediction is the complement of the sign bit of y.
        prediction := not_taken
    end if
    for j in 1..h in parallel do          Update the next h partial sums.
        k_j = h − j                       SR[k_j] is the partial sum for predicting the j^th next branch.
        if prediction = taken then
            SR'[k_j + 1] := SR[k_j] + W[i, j]    Do the next step in the perceptron output computation for the
        else                                     j^th next branch, speculating that this prediction is correct,
            SR'[k_j + 1] := SR[k_j] − W[i, j]    and shifting each partial sum to the next position. We're using
        end if                                   the current prediction as the j^th most recent history bit for
    end for                                      the j^th next branch.
    SR := SR'                             Copy the resulting partial sums into SR.
    SR[0] := 0                            Initialize the partial sum for h^th next branch.
    SG := (SG << 1) or prediction         Shift the prediction into the speculative global history.
end
```

Figure 4. Path-based neural prediction algorithm to predict branch at address pc

branches predicted modulo n. That is, $v[i]$ is the index of the row in W used for predicting the i^{th} most recent branch instruction. This array can be implemented as a small circular buffer global to all invocations of the training procedure with speculative and non-speculative versions as with the prediction algorithm. Note that the address modulo n was computed in the prediction algorithm, so it can be recorded in the circular buffer at that time. Also, the modulo operation need not be expensive: it is simply a masking operation if the number of weights vectors is chosen to be a power of two.

Some of the details of these algorithms have been omitted for clarity and brevity, e.g., details the maintenance of the circular buffer of weights vector indices and the maintenance of the contents of R, which is simply a non-speculative copy of the circuitry that maintains SR. A detailed Java implementation of the algorithm will be made available upon request.

3.2.4 Recovery After Misprediction

When the path-based neural predictor predicts incorrectly, the SR vector is restored to the value stored in R during the predictor update for the last committed branch. Since all of the branches up to the last committed branch were correctly predicted and committed in-order, the restored value of SR is as it was when the mispredicted branch was fetched, and prediction will continue normally. The recovery takes less than one cycle, and its latency is completely hidden by the latency of other actions taken by the microarchitecture to recover from the misprediction.

3.2.5 Area and Latency

Clearly, the prediction algorithm uses a slower method for computing y_{out} than the original perceptron method. However, since it begins the summation process h branches before the prediction is needed, *the latency is almost completely hidden*. The only elements on the critical path to making a prediction are reading the bias weight and adding it to the current partial sum (i.e., $SR[h]$). This is much faster than computing y_{out} all at once with a Wallace-tree and also consumes less area. The Wallace-tree for the original perceptron predictor has $O(h \log h)$ carry-save adders as well as a carry-lookahead adder for the final addition, while the new algorithm requires only $O(h)$ independent adders for updating SR at each prediction step. For reasonable-sized predictors and history lengths, we estimate that the path-based neural predictor would take approximately two clock cycles to produce a prediction given a branch address. This is the same latency tolerated by branch predictors from industrial designs [16]. We give details of these estimates later in Section 4.

4 Methodology

In this section, we describe our experimental methodology for evaluating the path-based neural predictor.

4.1 Microarchitectural Framework

We use 17 SPEC CPU integer benchmarks running under a version of SimpleScalar/Alpha [3], a cycle-accurate

procedure train (i, y_{out}: **integer**; *prediction* , *outcome* : { *taken* , *not_taken* },
v: **array** $[1..h]$ **of integer**; H: **array** $[1..h]$ **of** { *taken* , *not_taken* });
begin
if *prediction* $\neq$ *outcome* **or** $|y_{out}| \leq \theta$ **then** — *If incorrect or y_{out} below threshold then adjust weights*

$$W[i,0] := W[i,0] + \begin{cases} 1 & \text{if } outcome = taken \\ -1 & \text{if } outcome = not\_taken \end{cases}$$

for j **in** $1..h$ **in parallel do** — *k is the row in W whence came the j^{th} weight for*
$k := v[j]$ — *this prediction*

$$W[k,j] := W[k,j] + \begin{cases} 1 & \text{if } outcome = H[j] \\ -1 & \text{if } outcome \neq H[j] \end{cases}$$

Increment j^{th} weight for positive correlation,
end for — *decrement for negative correlation,*
end if
$G := (G << 1)$ **or** *outcome* — *Update non-speculative global history shift register*
if *prediction* $\neq$ *outcome* **then**
$SG := G$ — *Restore speculative history on a misprediction*
$SR := R$ — *Restore SR to a non-speculative version computed*
end if — *using only non-speculative information (not shown)*
end

Figure 5. Path-based predictor update algorithm

out-of-order execution simulator that has been enhanced to include our branch predictors, simulate overriding predictors at various latencies, and simulate deep pipelines. We simulate all of the SPEC CPU 2000 integer benchmarks, and all of the SPEC CPU 95 integer benchmarks that are not duplicated in SPEC CPU 2000. The benchmarks are compiled with the CompaQ GEM compiler with the optimization flags `-fast -O4 -arch ev6`.

To better capture the steady-state performance behavior of the programs, our experiments skip the first billion instructions, as several of the benchmarks have an initialization period lasting fewer than one billion instructions during which program behavior is not characteristic of the many billions of subsequent instructions. After skipping those instructions, each benchmark executes 500 million instructions on the `ref` inputs before the simulation ends.

Table 1 shows the base microarchitectural parameters used for the simulations. We started with a configuration loosely based on the Intel Pentium 4, with a deeper pipeline of 32 stages to provide a reasonable model of a future aggressively clocked microarchitecture. A recent study from Intel's Pentium Processor architecture group concludes that performance of aggressively clocked microarchitectures continues to improve until pipelines reach a depth of 52 [20]. Thus, while our 32-stage pipeline is aggressive for current technology, it is conservative with respect to what is possible in future technologies.

Parameter	Configuration
L1 I-cache	16 KB, 64B blocks, 2-way
L1 D-cache	8 KB, 64B blocks, 4-way
L2 unified cache	512KB, 128B blocks, 8-way
BTB	4096 entry, 2-way
Issue width	8
Pipeline depth	32
RUU entries	128
LSQ entries	128
L2 hit latency	7 cycles
L2 miss latency	200 cycles

Table 1. Microarchitectural parameters

4.2 Branch Predictors Simulated

We simulate the following predictors to compare with the path-based neural predictor:

2Bc-*gskew* We simulate a 2Bc-*gskew* predictor, which is a McFarling-style [13] hybrid predictor combining a bimodal predictor with an *egskew* predictor that predicts using the majority prediction of three components: the bimodal predictor and two *gshare*-like predictors indexed by special hash functions so as to minimize the chance that both predictors will suffer destructive interference at the same time. A version of this predictor would have been used in the Alpha EV8 processor [16]. In our latency-sensitive simulation, 2Bc-*gskew* takes more than one cycle to return a result. We use a two-level overriding organization [8] to mitigate this latency: A first-level 2K-entry bimodal predictor gives a prediction in a single cycle and instructions are fetched down the predicted path. If the second-level 2Bc-*gskew* predictor disagrees with the initial prediction, the instructions fetched so far are dropped and fetching continues from the other path. This technique closely reflects the design of the EV8 predictor, in which 2Bc-*gskew* overrides a less accurate instruction cache line predictor.

Perceptron Predictor We simulate a recent [10], highly accurate version of the perceptron predictor that combines global and per-branch history information in a manner reminiscent of the alloyed branch predictors of Skadron *et al.* [19, 10]. We again use an overriding organization with a first-level 2K-entry bimodal predictor, this time backed up with a second-level perceptron predictor. We note that this predictor has been shown to be more accurate than even the most aggressive multi-component hybrid predictor [10]. Thus, including other combined global and per-branch hybrid predictors in this study would be superfluous.

gshare.fast We simulate a specialized version of the *gshare* predictor that has been pipelined to return a result in a single cycle. By using older branch history to prefetch a portion of the pattern history table in a previous cycle and then using the exclusive-OR of more recent history and the low bits of the current branch address to select from that portion, *gshare.fast* has an effective latency of one cycle [7]. It has been shown to yield higher instruction per cycle rates than highly accurate predictors such as 2Bc-*gskew* and the perceptron predictor at large hardware budgets [7]. For this study, our simulation of *gshare.fast* is idealized, assuming that there is no overlap or missing gap between the older history and more recent history.

Fixed-Length Path Predictor We simulate a fixed length path branch predictor that forms a hash of the history of branch target addresses leading up to the branch to be predictor [21]. The hash function XORs the addresses, first rotating each address by a number of bits equal to it position in the branch history. The hash is used to index a table of two-bit saturating counters as in a two-level scheme. We use the same fixed length for each benchmark, as opposed to using a variable-length path branch predictor which requires expensive profiling [21]. (Note that none of the schemes used for this paper require profiling.)

Path-Based Neural Predictor We simulate the path-based neural predictor as described above, using an overriding organization with a first-level 2K-entry bimodal predictor as with the other overriding predictors.

Each simulated predictor is pipelined so that it can be accessed on every cycle, e.g. for a predictor with a latency of 2 cycles, the prediction requested 2 cycles ago is available in the current cycle. Each predictor's history registers are updated speculatively and corrected on a misprediction.

4.3 Tuning The Predictors

Using the `train` inputs of the benchmarks and trace-driven simulation, we find the history lengths that minimize the average misprediction rate for each hardware budget and branch predictor, exploring hardware budgets from 1 KB to 64KB. We use these history lengths in the execution-driven simulations on the `ref` inputs. Table 2 shows the tuned history lengths for each hardware budgets. Note that *gshare.fast* is not shown, as its history length is fully constrained by the details of its implementation, and is equal to the base-2 logarithm of the number of elements in the pattern history table.

Hardware Budget	fixed length path	2Bc-*gskew*	global/ local	path-based neural
1 KB	10	10	25/9	13
2 KB	10	10	31/11	18
4 KB	12	10	34/12	20
8 KB	15	11	34/12	32
16 KB	20	14	38/14	34
32 KB	20	15	40/14	34
64 KB	20	16	50/18	37

Table 2. Tuned history lengths

4.4 Estimating Branch Predictor Latency

We use CACTI 3.0 [18] to estimate the latency of the various memories accessed by the predictors. We use HSPICE along with a custom logic design program to estimate the latency of the circuits used to compute the perceptron output for the perceptron predictor as well as the latency of the adders used for the path-based neural predictor. Table 3 shows the latencies we derived for each branch predictor and hardware budget except for *gshare.fast*, giving the amount of time it takes from the time a branch address is known to the time a prediction becomes available. For *gshare.fast*, the latency is always at most one cycle. For 2Bc-*gskew*, we estimate the latency of the predictor as the delay in accessing the slowest table plus one fan-out-of-four (FO4) delay for taking the majority and choosing the hybrid prediction from the two component predictions. For the global/local perceptron predictor, the latency is the sum of the access delay to the table of weights vectors measured by CACTI and the worst-case delay of the perceptron output circuit as measured by HSPICE. We optimistically ignore the access time to the first-level table of per-branch histories. The fixed-length path branch predictor is computationally expensive to implement because it requires hashing many addresses to produce one prediction. Nevertheless, we optimistically assume that it can be pipelined to produce a result with the same latency as 2Bc-*gskew*. For the path-based neural predictor, the latency is the sum of the access delay to the table of bias weights and the worse-case delay of the adder that adds the bias weight to the next partial sum in the SR vector. For consistency, we use the same adder circuits that were used in the original perceptron predictor study [10]. All of the estimates assume a 90 nm technology and an aggressive 8 FO4 delays, i.e., 3.86 GHz.

Hardware Budget	2Bc-*gskew* (cycles)	global/local perceptron	path-based neural
1 KB	2	5	2
2 KB	2	5	2
4 KB	2	5	2
8 KB	2	6	2
16 KB	2	6	2
32 KB	2	6	2
64 KB	3	7	3

Table 3. Estimated access latencies

5 Experimental Results

In this section, we give the results of our experimental studies. We discuss the misprediction rates of the various branch predictors. We then discuss the performance achieved by the predictors in terms of instructions-per-cycle (IPC).

5.1 Misprediction Rates

Figure 6 shows the arithmetic mean misprediction rates for the four predictors ranging over hardware budgets from 1 KB to 64KB over all benchmarks as measured by the microarchitectural simulator. Clearly, the path-based neural predictor has the lowest misprediction rate of all the predictors for all hardware budgets. Figure 7 shows the misprediction rates for each benchmark at a 8KB hardware budget. The path-based neural predictor achieves an average misprediction rate of 5.7%, which is 7% lower than that of the global/local perceptron predictor at 6.1%, 13% lower than that of 2Bc-*gskew* at 6.6%, and 40% lower than that of the fixed-length path branch predictor at 9.4%. The path-based neural predictor has the lowest misprediction rate of all the predictors in 9 out of the 17 benchmarks. Ignoring the global/local perceptron predictor, the path-based neural predictor is the best predictor for 14 of the benchmarks.

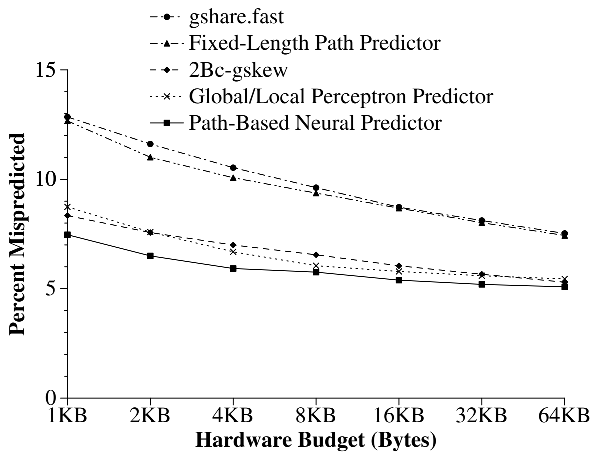

Figure 6. Average misprediction rates per hardware budget

5.2 Instructions Per Cycle

Figure 8 shows the number of instructions executed per cycle (IPC) for each branch predictor and hardware budget. Clearly, the path-based neural predictor yields the best performance at every hardware budget. The key reason is the combination of superior accuracy and low latency. For instance, the global/local perceptron predictor, which is the second most accurate of all the branch predictors, yields the worse performance at higher hardware budgets because of its high latency. At the same time, 2Bc-*gskew*, a McFarling-style hybrid with approximately the same latency as the path-based neural predictor, delivers less accuracy and performance than the single-component path-based neural predictor. At a 64KB hardware budget, the path-based neural predictor delivers an IPC 16% higher than that of the perceptron predictor because of that predictor's high latency.

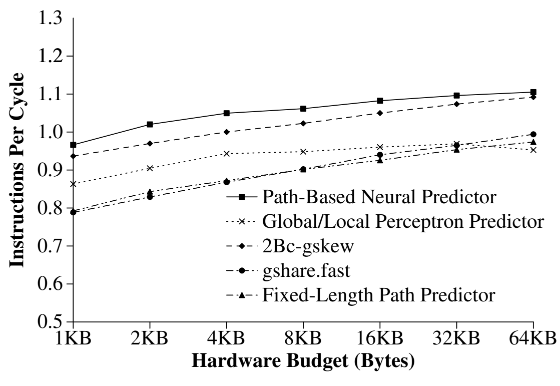

Figure 8. Average IPC per hardware budget

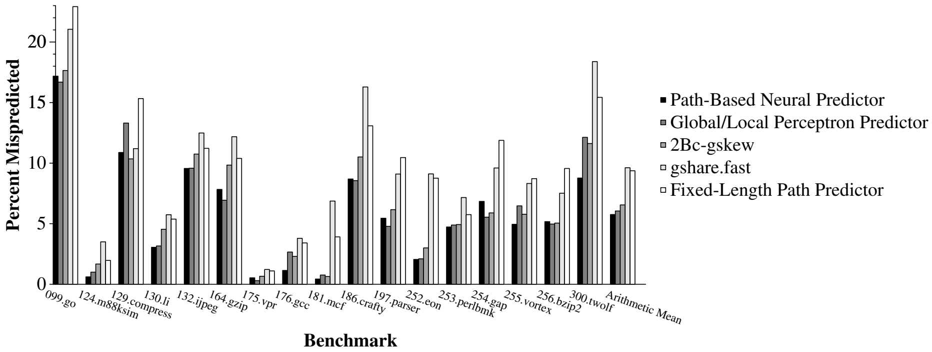

Figure 7. Misprediction rates per benchmark at an 8KB hardware budget

Figure 9 shows the IPC for each benchmark and each predictor at an 8KB hardware budget. The path-based neural predictor yields the best IPC in 15 out of the 17 benchmarks. It achieves a harmonic mean IPC of 1.06, giving a speedup of 12% over the global/local perceptron predictor at 0.95 IPC, 4% over 2Bc-*gskew* at 1.02 IPC, 18% over *gshare.fast* at 0.90 IPC, and 18% over the fixed length path branch predictor at 0.90 IPC. At this hardware budget, both 2Bc-*gskew* and the path-based neural predictor have a latency of 2 cycles, while *gshare.fast* has a single-cycle latency. The global/local perceptron predictor has a latency of 6 cycles at this hardware budget. Although it is more accurate than *gshare.fast* and 2Bc-*gskew*, its higher latency cancels any advantage it might have for performance.

5.2.1 Area vs. Hardware Budget

Although standard for branch prediction research, equating the term *hardware budget* with number of bits of predictor state is problematic in our case. As described in Section 3.2.3, an implementation of the path-based neural predictor may use $h + 1$ independently addressable memories, each with its own selection logic, to facilitate the update algorithm. The path-based neural predictor also requires a number of adder circuits proportional to the history length. We estimate that a naive implementation of a path-based neural predictor using 8KB of state could require 80% more area than a 8KB 2Bc-*gskew* predictor. Even so, the path-based neural predictor is still the best choice. A path-based neural predictor with a hardware budget of 4KB, consuming approximately 10% less total area than a 8KB 2Bc-*gskew*, achieves a harmonic mean IPC of 1.05 which is less than 1% lower than that of an 8KB path-based neural predictor and 3% higher than that of a 8KB 2Bc-*gskew*. Indeed, a path-based neural predictor with only 2KB of state achieves the same IPC as an 8KB 2Bc-*gskew*.

6 Conclusion

We have presented a new neural branch predictor that has lower latency and superior accuracy to previous neural branch predictors. Our new predictor achieves high accuracy and low latency by predicting a branch using a neuron selected dynamically along the path to that branch. This work is only the beginning of path-based neural prediction; we have yet to fully exploit the potential of this technique. We have shown that our predictor has better accuracy and yields higher performance than conventional predictors. By incorporating our path-based neural predictor into new microarchitectures, designers will be able to improve IPC rates while increasing pipeline depths and clock frequencies.

7 Acknowledgments

I thank Calvin Lin and Doug Burger for their helpful comments on the first draft of this paper. Thanks also to Charles Ganansia for working on circuit models for this research. The preparation of the final version of this paper is supported by NSF Grant CCR-0311091.

References

[1] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[2] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. Hierarchical scheduling windows. In *Proceedings of the 34th International Symposium on Microarchitecture*, Istanbul, Turkey, December 2002.

[3] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[4] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–92, June 1995.

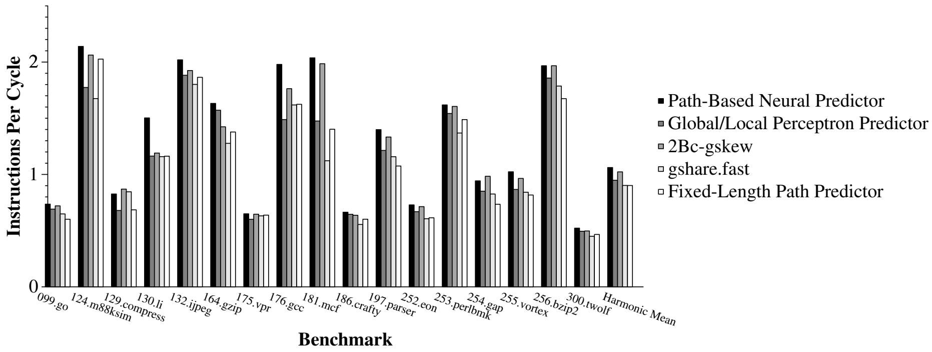

Figure 9. IPC per benchmark at an 8KB hardware budget

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[6] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, July 1998.

[7] Daniel A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th Int'l Symposium on High Performance Computer Architecture*, February 2002.

[8] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 67–76, December 2000.

[9] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th Int'l Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[10] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[11] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[12] Gabriel H. Loh and Dana S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 11th Conference on Parallel Architectures and Compilation Techniques*, pages 165–176, Charlottesville, Virginia, September 2002.

[13] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[14] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, December 1995.

[15] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

[16] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[17] André Seznec and Antony Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, California, June 2003.

[18] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[19] Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–206, October 2000.

[20] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, Anchorage, Alaska, May 2002.

[21] Jared Stark, Marius Evers, and Yale N. Patt. Variable length path branch prediction. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 170–179, October 1998.

[22] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, California, June 2003.

[23] Lucian N. Vintan and M. Iridon. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 868–873, July 1999.